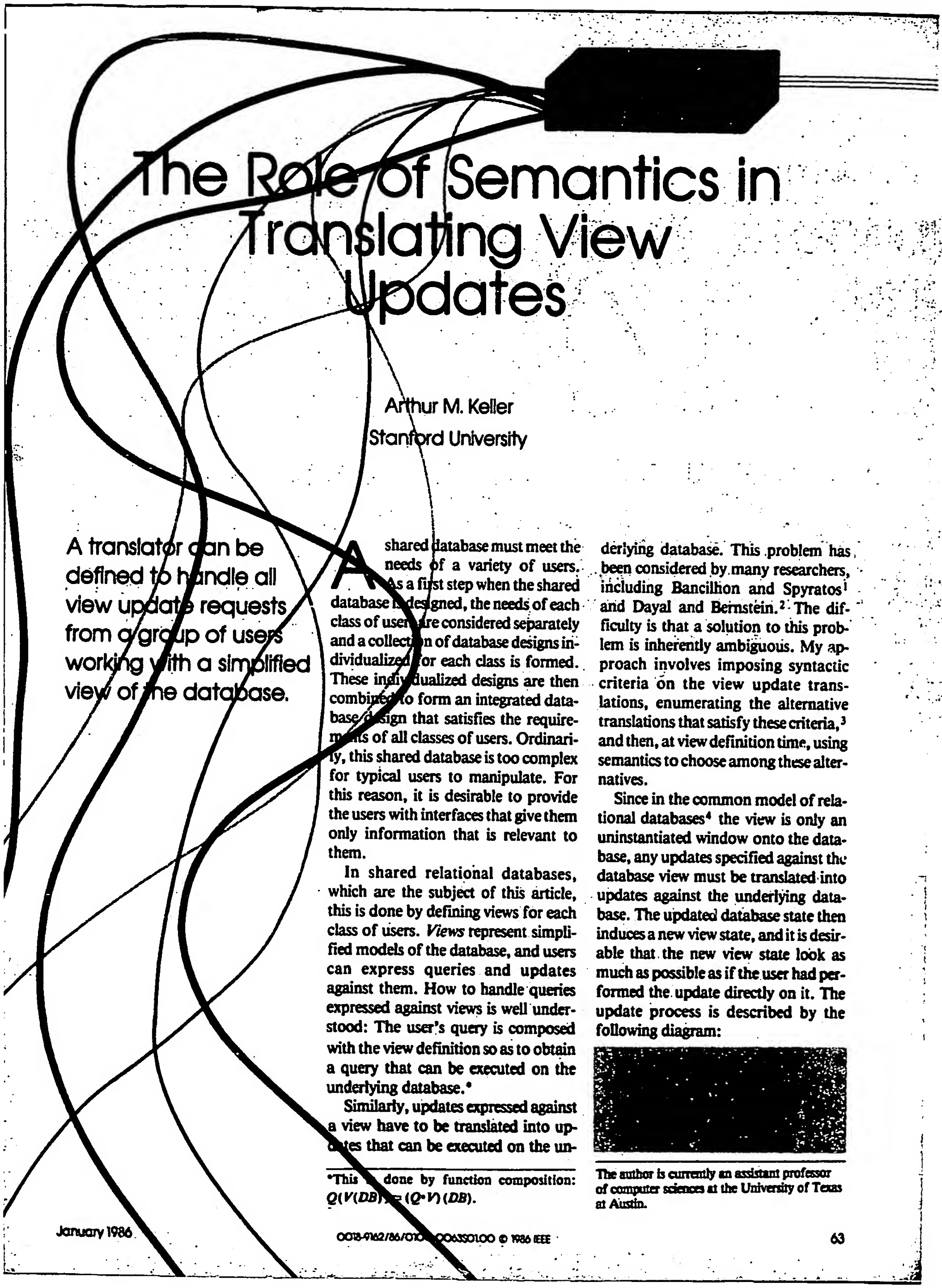


The Role of Semantics in Translating View Updates

Arthur M. Keller
Stanford University

A translator can be defined to handle all view update requests from a group of users working with a simplified view of the database.

A shared database must meet the needs of a variety of users. As a first step when the shared database is designed, the needs of each class of users are considered separately and a collection of database designs individualized for each class is formed. These individualized designs are then combined to form an integrated database design that satisfies the requirements of all classes of users. Ordinarily, this shared database is too complex for typical users to manipulate. For this reason, it is desirable to provide the users with interfaces that give them only information that is relevant to them.

In shared relational databases, which are the subject of this article, this is done by defining views for each class of users. *Views* represent simplified models of the database, and users can express queries and updates against them. How to handle queries expressed against views is well understood: The user's query is composed with the view definition so as to obtain a query that can be executed on the underlying database.*

Similarly, updates expressed against a view have to be translated into updates that can be executed on the underlying database. This problem has been considered by many researchers, including Bancilhon and Spyratos[1] and Dayal and Bernstein.[2] The difficulty is that a solution to this problem is inherently ambiguous. My approach involves imposing syntactic criteria on the view update translations, enumerating the alternative translations that satisfy these criteria,[3] and then, at view definition time, using semantics to choose among these alternatives.

Since in the common model of relational databases[4] the view is only an uninstantiated window onto the database, any updates specified against the database view must be translated into updates against the underlying database. The updated database state then induces a new view state, and it is desirable that the new view state look as much as possible as if the user had performed the update directly on it. The update process is described by the following diagram:

The author is currently an assistant professor of computer sciences at the University of Texas at Austin.

*This is done by function composition: $Q(V(DB)) = (Q \cdot V)(DB)$.

0018-9162/86/0100-0063$01.00 © 1986 IEEE

The initial database state, DB, is mapped by means of the view, V, into view state $V(DB)$. The user specifies update U against this view state to obtain new view state $U(V(DB))$. To have a permanent effect, the view update, U, must be translated into a database update sequence. The corresponding (translated) database update sequence, $T(U)$, is performed on the database to obtain database state $T(U)(DB)$, which we will also call DB'. With the new database state, DB', the new view state is $V(DB')$. We say that this translation from view update U to database update sequence $T(U)$ has *no side effects in the view* if $V(DB') = U(V(DB))$, that is, if the view has changed precisely in accordance with the user's request. Side effects are undesirable and are to be avoided whenever possible. In fact, no side effects occur in translating updates expressed against views that consist solely of the relational operators **select** and **project**. However, in some cases, updates expressed against views that involve **joins** (the operators that link relations in relational databases) cannot be translated unless some side effects are permitted.

The question is, given a view update U, what is the correct database update sequence $T(U)$? Our goal is to find a translator T that supplies a $T(U)$ for each valid U. This translator is a *functional*, since it maps from functions $[U]$ into other functions $[T(U)]$. It is based on the database definition and the view definition, as well as the semantics of the problem. Such a translator should be chosen in advance by an expert so that updates supplied by users can be performed without ambiguity. That is, we want to resolve any ambiguity completely through the choice of a view update translator at view definition time.

The choice of a translator depends on a characterization of the set of possible translators. We characterize this set of translators by enumerating the possible translations for each view update request. In the diagram above, database update sequence $T(U)$ is a translation of the view update request U. For each view update request U, the translator T chooses a database update sequence $T(U)$ to implement the request on the database. There may be zero translations, one translation, or many translations $T(U)$ corresponding to any given update request U, since the desired view state can correspond to many database states. If there is precisely one update translation, we should use that one. If there are many translations, we need to choose one. Translations that involve the simplest changes to the database are probably better than those that involve more complex changes to the database. Using criteria based on this idea, we can reduce this ambiguity, but not eliminate it. We can, however, enumerate the various translation alternatives.

On the other hand, if there are no translations $T(U)$ corresponding to a given update request, it is because there are no database states corresponding to the desired view state. In this case, the view state may not be achievable because the user's request implies additional changes that affect that view state. If we allow such additional changes, which are *side effects* in the view, there may be translations $T(U)$ that correspond to the elaborated view update U.

View update translation

Definitions. We need to define a few terms to explain the process of translation of view updates into database updates.[5,6] A *domain* is a finite set of values. A *relation schema* is a tagged set of domains and a set of constraints that tuples in the relation must satisfy. The tags are the *attribute* names of the relation. A *tuple* is a tagged set of values; one value is drawn from each respective domain. (In the example view and database diagrams below, tuples are read horizontally and the attributes are the columns.) The *extension of a relation* is the set of tuples in the relation.

Functional dependencies and key dependencies are examples of constraints. A *key dependency* indicates that there is at most one tuple with any given values for the attributes in the key. (A *key* can be thought of as a unique identifier for a particular record.) A *functional dependency* is a generalization of the concept of a key dependency: We say that the attributes in set X *functionally determine* the attributes in set Y (written $X \rightarrow Y$) if whenever two tuples have matching values for all attributes in X, they also have matching values for all attributes in Y.

A *database schema* is a set of relation schemata indexed by relation name. A *database extension* is a set of relation extensions; there is one database extension for each relation in the database schema.

A *database view definition* is a mapping whose domain is the set of all relation extensions for a given database schema. The *range* of a database view definition is also a set of relation extensions for a schema specific to the view definition. The mapping from the database that is the domain of that view to the relation in the range of the view is defined by a type of database query. The *view extension* is the result of the database query that defines the view when performed on the underlying database that is the domain of the view.

Views are defined by database queries. There are few restrictions on the nature of view definitions for views defined only for queries. But we must impose significant restrictions on views that we want to be able to update. For example, it is undesirable to update through views that include *aggregations.** Consider an employee

*An *n*-ary function f whose domain is a finite power set is monotonic if $(\forall)(R_i \subseteq S_i) \rightarrow f(R_1,\ldots,R_n) \subseteq f(S_1,\ldots,S_n)$. Select, project, join and union of relations are monotonic functions.[7] The set difference operator and aggregations are nonmonotonic. My theory does not support views that are nonmonotonic.

relation with Employee and Department attributes, and a Department view with Department and Number of employees attributes. The view update request to increment the number of employees in the Computer department would result in the need to add an unknown employee to the Employee relation. A request to decrement the number of employees in the Toy department would require the computer to choose an employee to fire—a decision best done by some person.

Employee relation

employee	department	salary
Joe	Computer	25,000
Sarah	Computer	30,000
George	Toy	20,000
Fred	Toy	25,000

Manager relation

department	manager
Computer	Sheryl
Toy	Sam

Department view

department	number of employees
Computer	2
Toy	2

Operators. The relational operators **select**, **project**, and **join** are more suitable than some of the other relational operators (for example, **divide** and **set difference**) for defining updatable views. The **select** operator extracts the tuples from a relation that satisfy the selection condition. The tuples that do not satisfy the selection condition do not appear in the view. For example, given the Employee relation shown above, we can define a view for the manager of the Toy department that selects only employees in the Toy department:

Toy department manager's view

employee	department	salary
George	Toy	20,000
Fred	Toy	25,000

The **project** operator extracts the desired attributes from each tuple in a relation. For example, we may want to make available a view of the Employee relation that does not include the salary attribute:

Sanitized employee relation

employee	department
Joe	Computer
Sarah	Computer
George	Toy
Fred	Toy

The **join** operator combines two relations, and in so doing, combines tuples with matching attribute values. If we take the join of the Employee and Manager relations above, we obtain the following view:

Employee-manager view

employee	department	salary	manager
Joe	Computer	25,000	Sheryl
Sarah	Computer	30,000	Sheryl
George	Toy	20,000	Sam
Fred	Toy	25,000	Sam

The **union** operator combines two or more relations with the same schema definition into a single relation with the same schema definition. The **union** operator is commonly used in distributed databases to combine the partitioned components of the relation. For example, consider a company with three locations, Boston, Palo Alto, and Austin. It has employee records containing name, department, and location, which are stored partitioned by location. A corporate manager might be given the combined view (the "Corporate employee view," below) that is the union of the three location-defined employee relations, which follow:

Austin employee relation

employee	department	location
Joe	Computer	Austin
Sarah	Computer	Austin

Boston employee relation

employee	department	location
George	Toy	Boston
Fred	Toy	Boston

Palo Alto employee relation

employee	department	location
Don	Computer	Palo Alto
Allan	Housing	Palo Alto
Maureen	Housing	Palo Alto

Corporate employee view

employee	department	location
Joe	Computer	Austin
Sarah	Computer	Austin
George	Toy	Boston
Fred	Toy	Boston
Don	Computer	Palo Alto
Allan	Housing	Palo Alto
Maureen	Housing	Palo Alto

The **select**, **project**, **join**, and **union** operators have the interesting relationship illustrated by the following diagram. **Select** and **project** extract information from a single relation. **Union** and **join** combine information from more than one relation to form a single relation. **Select** and **union** are horizontal operators that refer to collections of tuples. **Project** and **join** are vertical operators that refer to attributes.

	horizontal	vertical
extracting	selection	projection
combining	union	join

Constraints on views and databases. The constraints on the database and view are as important for view update translation as the operators that define the view. The constraints I discuss in this article are *key dependencies* (in which each tuple in a relation has a unique key) and *inclusion dependencies* on **join** fields (such as the specification that each employee must be in a department that exists in the Department relation).

Other dependencies can also cause problems in view update translation. An example of such dependencies is an *explicit functional dependency*.[8] In an explicit functional dependency, one or more of the attributes in a tuple can be computed based on the values of some other attributes in that tuple. One seemingly innocuous explicit functional dependency holds in the following relation:

```
Orders relation
item | price | quantity | total
-----+-------+----------+------
6    | 10    | 7        | 70
5    | 8     | 20       | 160
10   | 15    | 20       | 300
```

In the Orders relation shown above, the price times the quantity equals the total. The view containing item, price, and quantity fields (but not the final field—total) appears below:

```
Orders view
item | price | quantity
-----+-------+---------
5    | 10    | 7
6    | 8     | 20
10   | 15    | 20
```

A change to quantity or price implies some change to the total for that tuple. My approach does not handle constraints that, unlike functional dependencies, can be open ended and arbitrarily difficult to compute. For example, it is possible in some approaches to imbed NP-complete, or incomputable, problems as explicit functional dependencies. I require instead that any combination of values forming the attributes for a tuple in any one relation must be acceptable in some database state that satisfies the constraints for the schema under consideration. This restriction is stronger than merely eliminating explicit functional dependencies.

A *database update* can be made directly against the database, provided it satisfies the constraints on the database. A *view update* is merely an update that is described against the view, but it must be translated into a sequence of database updates in order to be executed. There may be several candidate sequences of database updates corresponding to one view update. We call these sequences of database updates the *translations* of the view update request. We say that a translation is *valid* if it performs the view update as requested.

For updates through select and project views, we require that the new view extension be precisely the result of performing the view update on the old view extension, as if the view in question were an ordinary relation. It may not be possible to perform updates through views that include joins without additional changes occurring in the view.[9] These *view side effects* are a result of functional dependencies that require that changes requested in the view tuples be consistent with the remainder of the database. The underlying tuples corresponding to a view tuple are the database tuples with keys matching the keys that appear in the view tuple. The side effects occur if view tuples share corresponding underlying tuples that undergo updates.

Requiring that a translation be valid is not sufficient for our purposes—it is only a first step. I have defined five additional criteria that the translations must satisfy.[3] These criteria proscribe (1) database side effects, (2) multiple changes to the same database tuple, (3) unnecessary database changes, (4) replacements that can be simplified, and (5) delete-insert pairs on the same relation. The criteria are used to obtain only the simplest (or a minimal number of) view update translations.

A *view update translator* is a mapping from view update requests into translations of these view update requests. A *translator* takes the user's view update requests and translates them into database update requests that can be processed by the database system. Thus, a view update translation facility would be a useful adjunct to a database system that has a view definition facility.

Various approaches to the view update problem

Furtado and Casanova[10] provide a good overview of the view update problem. They do not propose any new solutions, but they do clearly explain previous approaches. They complain that "a common tendency is to define views only for query and authorization purposes, and afterwards try to solve the problems created by view updates. [They] feel that the update requirements should be considered from the start...."[10]

Dayal and Bernstein[2] provide a strong theoretical foundation for work on this problem. They provide some algorithms for translating a restricted class of view updates. They define a unique translator for each view definition, and the translator is based solely on structural considerations. The section below titled "Examples of update operations commonly performed on databases and views" shows how structural considerations alone are insufficient to choose a view update translator.

Davidson and Kaplan[11,12] consider the related problem of natural language updates to databases. Their approach involves an incomplete enumeration of translations and uses heuristics to choose one of these translations. My approach involves a complete enumeration of translations and uses semantics supplied at view definition time to choose a translator for that view definition that will handle all view updates.

Bancilhon and Spyratos[1] prove a mathematically elegant relationship between view update translators and constant view complements. Two views are *complementary* if together they supply sufficient information to determine the entire database state. Bancilhon and Spyratos show that a unique view update translator exists for a given view when a complementary view is chosen to be held constant. This notion, however, is contrary to the notion of data sharing. Furthermore, the approach that they developed based on this relationship has significant practical drawbacks.[7]

Cosmadakis and Papadimitriou[8] base their work on Bancilhon and Spyratos's[1] notion of constant view complements. Their work primarily involves analysis, not general algorithms for view update translation. They show how difficult it is to use complements. (For example, they show that finding a minimal complement of a given view is NP-complete.) They adopt a *universal relation assumption*[13]; that is, they see views as essentially projections of a given relation. The implication of their work is that an approach different from constant complements is needed.

Hegner's approach,[14] also based on constant view complements of Bancilhon and Spyratos, attempts to eliminate the ambiguity by choosing particular complements, called "components."

Carlson and Arora[15] ignore **replacement** requests by claiming that they can be decomposed into a sequence of **insertions** and **deletions**. I believe that **replacement** requests are different from **deletion** and **insertion** requests, since they do not require a consistent database state between the two operations.

Furtado, Sevcik, and dos Santos[16] propose using the **union** operator for defining views and claim that **insertions**, **deletions**, and **replacements** specified against a view defined by a **union** should be applied to each of the underlying merged relations in the database. The **union** operator is useful for distributed databases where a relation is partitioned over several sites; however, where the relations are merged by the **union** operator, an insertion must occur at only one site and cannot be performed, as the authors suggest, at all sites.

Clemons,[17] Sevcik and Furtado,[18] Rowe and Schoens,[19] and Tuchermann, Furtado, and Casanova[20] all use an abstract data type approach to the view update problem.

Masunaga[21] proposes the use of semantics for resolving ambiguity in the view update problem. He does not enumerate all possible translations, but rather an incomplete set of translations—those that handle his class of updates. (He does not describe how these semantics are obtained or exactly how they are used.) He also suggests that some ambiguity can be resolved by having the system ask questions of the user when the view update is requested. I suggest that all ambiguity be completely resolved by obtaining the semantics at view definition time from the view definer, typically the database administrator, so that no ambiguity remains when users provide view updates.

Examples of update operations commonly performed on databases and views

The update operations on databases and views that I consider below are **deletion**, **insertion**, and **replacement**. A **deletion** is the removal of a single tuple from a relation. An **insertion** is the addition of a single tuple to a relation. A **replacement** is the combination of a **deletion** from and an **insertion** into the same relation in a single, atomic action that does not require a consistent intermediate state between the **deletion** and **insertion** steps. An update is always either a **deletion**, an **insertion**, or a **replacement**.

Let us consider an Employee relation that contains each employee's name and location and tells whether the employee is a member of the company baseball team. The company has two locations: New York and Austin. Baseball team members must be employees. The database state follows:

[illegible]

[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]

The personnel manager in New York, Susan, has the following view definition:

```
[illegible]
```

Her view state follows:

[illegible]

[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]

Suppose she requests the deletion of "Joe" from her view. A reasonable translation of this request is to delete Joe's employee record from the underlying database, that is, to translate a **view deletion** into a **database deletion**. If the employee was a member of the baseball team, he is thus removed from that membership as well.

The baseball team manager, Frank, has the following view definition:

```
SELECT *
FROM EMP
WHERE Baseball = "Yes"
```

His view state follows:

[illegible]

[illegible]	[illegible]	[illegible]
Frank	[illegible]	[illegible]
Joe	[illegible]	[illegible]
Sally	Austin	Yes

Suppose he requests deletion of "Sally" from his view. It is unreasonable to delete the "Sally" employee tuple from the underlying database (unless one believes that baseball is all important). A reasonable translation of this view deletion request is to replace the Baseball attribute of the underlying database tuple with a "No," thus translating a **view deletion** into a **database replacement**.

One might argue that Frank's **view deletion** request should have been made in the form of a replacement. However, this would mean requesting the replacement of a tuple (the "Sally" tuple) in the view with a view tuple that does not appear in the view. Then Frank's request would not be valid in the view, as the replacement tuple could not possibly be a view tuple. In addition, Frank would have to make a distinction between deletion and replacement that he cannot make by looking at the effects that updates have on his view.

Let us compare the replacement and deletion operations further. It is possible to translate Susan's request by changing Joe's location value in the Employee relation to Austin. (I doubt that the Austin manager would be pleased by such an implementation.) A request that, like Susan's, could change the underlying database in more than one way should be issued as a replacement request by someone authorized to access the entire relation. That person would need access to a view that permits seeing all the effects of such a request.

We see that a view deletion request is sometimes best translated into a database deletion request and at other times into a database replacement request. Similar alternatives arise for insertion requests, which can be translated into database insertion requests and database replacement requests.

The two views on the Employee relation (Susan's and Frank's) have the same syntactic structure, but differing semantic structures. Therefore, we can conclude that semantic considerations alone are insufficient to disambiguate the view update problem. Let us consider some of the semantics not captured in the database and view definitions shown above that might be useful in view update translation.

While the company may currently have two locations, New York and Austin, there are many other locations where the company could open up offices. Thus, the location value "San Francisco" might be selected from among many possible location values. To remove an employee from the New York office means a corresponding database change achieved by either deleting the employee record or changing the location in the record to Austin or San Francisco. In instances where there are many possible locations, changing location values in the database to reflect employee relocation involves choosing one of them.

On the other hand, the semantics of the baseball team are that employees either are players who play various positions, such as pitcher, catcher, or first baseman, or sit on the bench waiting to play—or they are not on the team at all. In this case, the values of the Baseball team field would be positions, not "Yes" or "No." In formulating a view describing the baseball team (like Frank's, shown above) records with a single value, that of not being on the team, are excluded from the view. Removing a record from the baseball team view means changing the database by either deleting the Employee record or changing the Baseball team member field. Since there is only one value of the Baseball team member field, "No" (that is, not a member of the team), that does not appear in the view, changing that field does not require choice of a particular value. The import of this observation is that a replacement of a database tuple in response to a view tuple deletion is more likely to be desired when there is a single excluded value for an attribute mentioned in the Select clause of the view definition.

We can use Susan's and Frank's views to illustrate the nature of interaction between users of different views. When Susan deletes Joe from her view, Joe also is removed from Frank's view. When Frank deletes Joe from the baseball team (and the original database state is thus changed), Joe's record in Susan's view is changed: He is shown as no longer on the baseball team. Such effects on other views may be somewhat disconcerting, but the same effects would occur if the changes were requested on the underlying database instead of the view. Since all operations on views are translated into operations on the database, a solution to the concurrency control problem on the database works for the concurrency control on the view, although it does not handle the problem of stale data appearing on a user's screen.

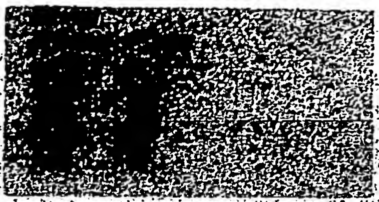
The purpose of a shared database is to facilitate controlled interaction between users. I do not consider the problem of interaction between multiple users—or, therefore, multiple views—in this article because this problem is quadratic in the number of views. Instead, I consider the interaction between each view and the underlying database, a problem that is only linear in the number of views. Understanding the interaction between each view and the underlying database naturally leads to an understanding of the interaction between views, but elaboration on the subject is beyond the scope of this article.

Examples of joins. A further level of ambiguity arises when views are constructed with join operations. Consider the following relations, where "employee" is the key of the ED relation and "department" is the key of the DM relation, the EDM view is formed by the natural join between the ED and DM relations, and the EM view is also formed by the natural join between the ED and DM relations, with the difference that the Department attribute is removed by a projection.

ED relation

employee	department
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]

[illegible]



Let us consider the request to replace <George,Music,Ira> in view EDM by <George,Music,Edo>. This request requires replacing <Music,Ira> in relation DM with <Music,Edo>. This necessarily has the side effect of changing <Fred, Music,Ira> to <Fred,Music,Edo> as a result of the functional dependency. If we refuse to permit the side effect, we cannot accept the update. The presence of the **join** attribute in the view update request makes the user's intention clear: The user wishes to change George's manager in the Music department. It would be reasonable to ask the user to confirm that the intention is also to change the manager of other employees in the Music department—in particular, Fred. Confirmation affects the user interface and not the algorithms for view update translation, so it is beyond the scope of this article.

Let us consider some **insertion** requests and their consequences. Consider the request to insert <Rita, Sally> into view EM. Is the desired department Art or Books? Consider the request to insert <Rita,Ira> into view EM. We can assume that the department is Music, but this is not *necessarily* so. Consider the request to insert <Rita,Boris> into view EM. We have no idea what department to use, since Boris is not already a manager.

Let us consider some **replacement** requests and their consequences. Consider the request to replace <George, Ira> in view EM by <George, Sally>. Should George be moved into the Art or the Books department, or should Sally become the manager of Music as well as Art and Books? (The latter alteration has the side effect of changing Fred's manager as well.) Consider the request to replace <George,Ira> in the same view by <George,Boris>. We could make Boris the manager of the Music department, which would have the side effect of changing Fred's manager, or we could move George into a new department and make Boris its manager, but we would have no idea what to call this new department.

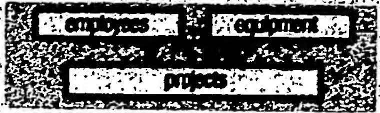
When the **join** attributes are removed from a view by means of **projections**, the view update problem becomes data dependent. My algorithms are data independent; the view definer decides what updates are to do in advance without consideration of the data, although the data is used to determine whether the update will succeed or fail.

When the key of a view is entirely contained in one underlying database relation, that relation is the *root relation*. If the key of the view is not entirely contained in one such relation, the fewest relations that contain the key are the *root relations*. Views that include **joins** are more easily updated when there is a single root relation.

Let me first illustrate a view that has multiple root relations, and then show how to change this view so that it has only one root relation.

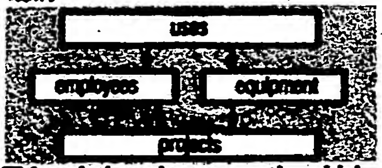
Consider the following example, which involves a query graph with two roots. We have three relations: Employees, Projects, and Equipment. There is a many-to-one type of join, a *reference connection*[22] from the Project attribute of the Employees relation to the Name attribute of the Projects relation, and also a reference connection from the Project attribute of the Equipment relation to the Name attribute of the Projects relation. The view is defined by the following:

This view definition can be described by the following query graph. Each node represents a relation, and each directed edge (indicated by the symbol "↑") represents a join from the many-to-one direction. This type of join is called an *extension join*,[23] and it arises naturally from the database design process, where it is called a *reference connection*.[22]



Suppose that we want to delete the view tuple asserting that employee Smith from the Widget Design project has a personal computer at his disposal. We could delete the record of either the employee or of the personal computer. If we delete the Smith tuple in the Employees relation, not only will the view tuple disappear, but also all other view tuples mentioning Smith—an undesirable and arbitrary side effect. Similarly, deleting the tuple for Smith's personal computer in the Equipment relation will cause all tuples mentioning that personal computer to disappear. We could delete both the employee and equipment tuples in question (an approach suggested by its symmetry), but this would cause any view tuple that mentions either Smith or the personal computer to be removed—an even more undesirable side effect.

In the following example, another relation, Uses, has been added to the view:



This relation shows exactly which employees use which pieces of equipment. Now we can delete the view tuple asserting that employee Smith from the Widget Design project has a personal computer at his disposal, which is done by deleting the corresponding tuple in the Uses relation in the database.

These two examples illustrate why we only update through views that have a single root relation.

EM view

employee	manager
Joe	Sally
Sarah	Sally
George	Ira
Fred	Ira

Let us consider the request to replace <George,Music,Ira> in view EDM by <Geoige,Music,Edo>. This request requires replacing <Music,Ira> in relation DM with <Music,Edo>. This necessarily has the side effect of changing <Fred, Music,Ira> to <Fred,Music,Edo> as a result of the functional dependency. If we refuse to permit the side effect, we cannot accept the update. The presence of the **join** attribute in the view update request makes the user's intention clear: The user wishes to change George's manager in the Music department. It would be reasonable to ask the user to confirm that the intention is also to change the manager of other employees in the Music department—in particular, Fred. Confirmation affects the user interface and not the algorithms for view update translation, so it is beyond the scope of this article.

Let us consider some **insertion** requests and their consequences. Consider the request to insert <Rita, Sally> into view EM. Is the desired department Art or Books? Consider the request to insert <Rita,Ira> into view EM. We can assume that the department is Music, but this is not *necessarily* so. Consider the request to insert <Rita,Boris> into view EM. We have no idea what department to use, since Boris is not already a manager.

Let us consider some **replacement** requests and their consequences. Consider the request to replace <George, Ira>in view EM by <George, Sally>. Should George be moved into the Art or the Books department, or should Sally become the manager of Music as well as Art and Books? (The latter alteration has the side effect of changing Fred's manager as well.) Consider the request to replace <George,Ira> in the same view by <George,Boris>. We could make Boris the manager of the Music department, which would have the side effect of changing Fred's manager, or we could move George into a new department and make Boris its manager, but we would have no idea what to call this new department.

When the **join** attributes are removed from a view by means of projections, the view update problem becomes data dependent. My algorithms are data independent; the view definer decides what updates are to do in advance without consideration of the data, although the data is used to determine whether the update will succeed or fail.

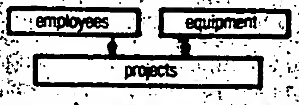
When the key of a view is entirely contained in one underlying database relation, that relation is the *root relation*. If the key of the view is not entirely contained in one such relation, the fewest relations that contain the key are the *root relations*. Views that include **joins** are more easily updated when there is a single root relation.

Let me first illustrate a view that has multiple root relations, and then show how to change this view so that it has only one root relation.

Consider the following example, which involves a query graph with two roots. We have three relations: Employees, Projects, and Equipment. There is a many-to-one type of join, a *reference connection*[22] from the Project attribute of the Employees relation to the Name attribute of the Projects relation, and also a reference connection from the Project attribute of the Equipment relation to the Name attribute of the Projects relation. The view is defined by the following:

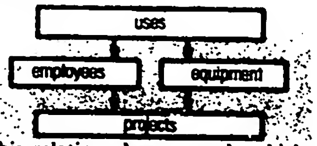
```
[illegible] AS
[illegible]
```

This view definition can be described by the following query graph. Each node represents a relation, and each directed edge (indicated by the symbol "|") represents a join from the many-to-one direction. This type of join is called an *extension join*,[23] and it arises naturally from the database design process, where it is called a *reference connection*.[22]



Suppose that we want to delete the view tuple asserting that employee Smith from the Widget Design project has a personal computer at his disposal. We could delete the record of either the employee or of the personal computer. If we delete the Smith tuple in the Employees relation, not only will the view tuple disappear, but also all other view tuples mentioning Smith—an undesirable and arbitrary side effect. Similarly, deleting the tuple for Smith's personal computer in the Equipment relation will cause all tuples mentioning that personal computer to disappear. We could delete both the employee and equipment tuples in question (an approach suggested by its symmetry), but this would cause any view tuple that mentions either Smith or the personal computer to be removed—an even more undesirable side effect.

In the following example, another relation, Uses, has been added to the view:



This relation shows exactly which employees use which pieces of equipment. Now we can delete the view tuple asserting that employee Smith from the Widget Design project has a personal computer at his disposal, which is done by deleting the corresponding tuple in the Uses relation in the database.

These two examples illustrate why we only update through views that have a single root relation.

Dialogue at view definition time

To reiterate my design for updating databases through views: I propose that the semantics necessary for disambiguating view update translation be obtained at view definition time. The semantics should then be used to choose a view update translator. Once a translator is chosen, users specify updates through the view, and the translator converts these view updates into database updates without any additional disambiguating dialogue. I have previously described the class of views and the view update translation that my approach supports.[3]

In the discussion that follows, I assume that the view is defined by a database administrator (DBA) who, by making use of a view-definition facility, will also provide the necessary semantics to choose a translator. While the effort that the DBA will need to make is the simplest case for the use of a view definition facility, it is clear that this system can be employed by any user with the wherewithal to define a view, either for the user's own use or for the use of other, perhaps less knowledgeable users. I regard the effort of collecting the semantics at view definition time as being amortized by their subsequent use in many view updates.

The candidate translators are organized conceptually into a tree, wherein each node represents a decision to be made. The semantics are merely the sequence of decisions made by the DBA in a walk of this tree guided by the view definition facility. The view definition facility presents questions to the DBA, each time supplying several options based on the view definition, the database schema, and the answers to the previous questions. Note that the tree of translators is different from the query graph representing the view. Note furthermore that the tree of translators is merely a pedagogical device; it does not actually exist within the view definition facility.

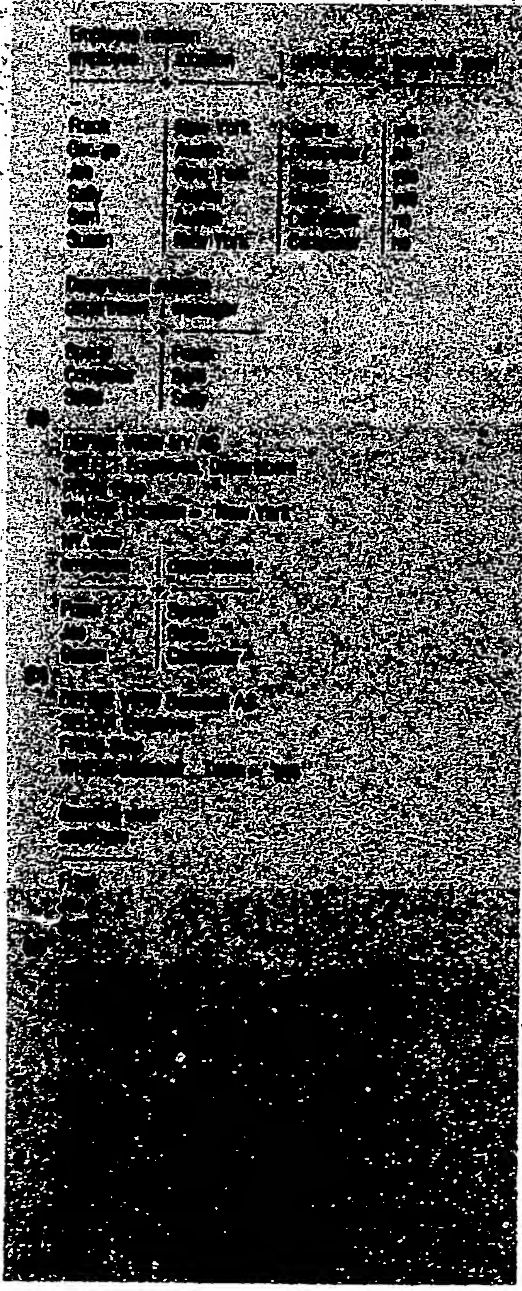

[illegible]

Figure 1. A database with the relations "Emp." and "Dept." (a), for which the views "NY" (b), "Baseball" (c), and "Manages" (d) are defined.

Choosing a translator does not involve using any information about the transactions that will be performed against the view. This is because the translator chosen will take as input individual view tuple updates and translate them into sets of database updates. Any information contained in the nature of the transactions performed that is useful for determining how to translate the update is captured during the view definition dialogue. Since the set of transactions is not necessarily available at view definition time, does not contain all the information needed for choosing a view update translator, and at best provides the same information that is obtained during the dialogue, I have chosen to use the dialogue to choose the translator instead. The algorithms that drive the dialogue are fully described elsewhere.[24]

The next three subsections illustrate the dialogue for defining a view update translator by arriving at translators for the three views ("NY," "Baseball," and "Manages") given in Figure 1. In that figure, a database state containing the relations "Emp." and "Dept." is followed by three view definitions, each of which is followed by the corresponding view state.

Dialogue at view definition: deletion. Stylized examples of the dialogue are given below for each of the three views. In each dialogue, the lines starting with "VDF" indicate questions asked by the view definition facility, and the lines starting with "DBA" indicate answers given by database administrator.

NY view

VDF: Is deletion of view tuples allowed?
DBA: Yes.
VDF: Does deletion of a view tuple mean deletion of the root database tuple, or replacement of that tuple?
DBA: Deletion.
VDF: OK, deletion of view tuples is now defined. Deletion of a view tuple means deletion of the corresponding database tuple.

Baseball view

VDF: Is deletion of view tuples allowed?
DBA: Yes.
VDF: Does deletion of a view tuple mean deletion of the root database tuple, or replacement of that tuple?
DBA: Replacement.
VDF: I assume that the baseball team field in the database tuple should be changed from "yes" to "no" when a view tuple is deleted. Is that correct?
DBA: Yes.
VDF: OK, deletion of view tuples is now defined. Deletion of a view tuple means replacing the corresponding database tuple by changing the baseball team field from "yes" to "no."

Manages view

VDF: Is deletion of view tuples allowed?
DBA: Yes.
VDF: OK, deletion of view tuples is now defined. Deletion of a view tuple means deletion of the corresponding tuple from the Employee relation.

Dialogue at view definition: insertion.

NY view

VDF: Is insertion of view tuples allowed?
DBA: Yes.
VDF: If a new tuple is inserted with an employee name matching some employee not in New York, should the employee location be changed to "New York," or should the request be rejected?
DBA: Rejected.
VDF: Can insertion of new view tuples cause insertion of new database tuples?
DBA: Yes.
VDF: On inserted tuples, what value should the baseball team have?
DBA: "No."
VDF: OK, insertion of view tuples is now defined. Insertion of a view tuple means insertion in the corresponding database tuple of location "New York" and baseball team "no."

Baseball view

VDF: Is insertion of view tuples allowed?
DBA: Yes.
VDF: If a new tuple is inserted with an employee name matching some employee not on the baseball team, should be employee's baseball team field be changed to "yes," or should the request be rejected?
DBA: Change.
VDF: Can insertion of new view tuples cause insertion of new database tuples?
DBA: No.
VDF: OK, insertion of view tuples is now defined. Insertion of a view tuple means replacement of the corresponding Employee relation tuple by changing baseball team to "yes." Only employees in Employee relation but not appearing in view may be inserted

Manages view

VDF: Is insertion of view tuples allowed?
DBA: Yes.
VDF: Can insertion of new view tuples cause insertion of new database tuples in the Employee relation?
DBA: Yes.
VDF: On inserted tuples, what value should the baseball team have?
DBA: "No."
VDF: Can insertion of new view tuples cause insertion of new database tuples in the Department relation?
DBA: Yes.
VDF: Can insertion of new view tuples cause replacement of existing tuples in the Department relation, for example, if the manager field differs?
DBA: No.
VDF: OK, insertion of view tuples is now defined. Insertion of a view tuple means insertion of the corresponding Employee relation tuple with baseball team value of "no" and, for a new department, insertion of the corresponding Department tuple. The update will be rejected if a new employee is added with a different manager from the one already listed for an existing department.

Dialog at view definition: replacement.

NY view

VDF: Is replacement of view tuples allowed?
DBA: Yes.

VDF: Can employee name be changed?
DBA: No.
VDF: Can department name be changed?
DBA: Yes.
VDF: OK, replacement of view tuples is now defined. Replacement of department name will change the department name in the corresponding Employee relation tuple.

Baseball view

VDF: Is replacement of view tuples allowed?
DBA: Yes.
VDF: Can employee name be changed?
DBA: Yes.
VDF: I assume that the old employee has baseball team changed to "no." Is that correct?
DBA: Yes.
VDF: I assume that the new employee must be in Employee relation and has baseball team changed to "yes."
DBA: Yes.
VDF: OK, replacement of view tuples is now defined. Replacement of employee name will change the database by changing the old employee's baseball team to "no" and the new employee's baseball team to "yes." Only employees in the Employee relation but not appearing in view may be replacements.

Manages view

VDF: Is replacement of view tuples allowed?
DBA: Yes.
VDF: Can employee name be changed?
DBA: No.
VDF: Can department name be changed for an existing employee?
DBA: Yes.
VDF: Can a new department be created in Department relation?
DBA: Yes.
VDF: Can manager be changed for an existing department?
DBA: Yes.
VDF: OK, replacement of view tuples is now defined. Replacement of a view tuple will change the corresponding Employee relation tuple if the department field changes, will insert a Department relation tuple for a new department, or will replace an existing Department relation tuple in the view update if the manager differs from the manager of an existing department.

I have described a method that can make updating relational databases through views reliable and convenient. The database administrator (DBA) defines the view and answers a sequence of questions to choose a valid view update translator for a large class of select, project, and join views. The definition of the translator is stored along with the view definition. The class of translators to choose from is based on the algorithm templates that generate all possible translations that satisfy five view-update-translation criteria.[3]

After the view and translator are defined, users can request insertions, deletions, and replacements through the view, and these are translated by the chosen translator into database updates, without the necessity for any disambiguating dialogue. Side effects can result from some insertions and replacements if permitted by the DBA; it may be desirable to have the user confirm such side effects, especially for insertions.

Not all possible translators will be captured by the answers to the DBA's questions. The set of candidate view update translators is quite large; in an earlier work,[3] I characterized this set by enumerating the set of all view update translations. Some of the translators that will result from the questioning will translate all updates that have translations satisfying the user's criteria; others will reject some updates because they were proscribed by the answers given by the DBA to the questions asked by the view definition facility. The translators resulting from the dialogue are simple; in fact, they were used to generate the enumeration of translations.

The process of defining a view and choosing a translator has been described here as being performed by the DBA. While the amount of work the DBA will need to perform is the simplest case for the use of a view update facility, it is clear that my system can be used by anyone with the wherewithal to define a view. The distinction to make is that a dialogue used to select a view update translator is most effective for static views that are defined once and used repeatedly. For dynamic views defined by natural language dialogue or universal relation interfaces, the overhead of answering the questions is not amortized over time by performing many view updates. In such cases, heuristics and user profiles can be used to determine choice of translator.[8]

Querying and updating through a view reduces the problems of security and protection, but does not eliminate them. Clearly, a view circumscribes the collection of data a user is permitted to access. The question of how to give each manager access to the data for his or her department can be addressed either by parameterizing the view to show only that department's data or by a parameterized protection scheme that allows the manager access only to tuples containing data for that department. Using both may seem redundant, but need not be. A parameterized view will make fewer demands on the database and the security system. A security system would have a large loophole if it were to give special consideration to queries and updates specified through views. Of course, an effective security system is needed when a view definition facility and an *ad hoc* query facility are made available to users.

With views and queries that are described non-procedurally, relational databases are an effective tool for productivity.[25] I have shown how to describe view update translators non-procedurally by answering a sequence of questions based on the view definition and the database structure. This non-procedural description facility has the potential to dramatically increase the productivity of views, and consequently, of relational databases. □

Acknowledgments

I thank Gio Wiederhold, Jeff Ullmann, Christos Papadimitriou, and Richard Shuey for their advice and encouragement.

This work was supported in part by Contract N39-84-C-0211 (the Knowledge Base Management Systems Project, Professor Gio Wiederhold, principal investigator) from the Defense Advanced Research Projects Agency (DARPA). The views and conclusions contained in it are those of the author and should not be interpreted as representative of the official policies of DARPA or the US Government.

References

1. F. Bancilhon and N. Spyratos, "Update Semantics and Relational Views," *ACM Trans. Database Systems*, Vol. 6, No. 4, Dec. 1981.
2. U. Dayal and P. A. Bernstein, "On the Correct Translation of Update Operations on Relational Views," *ACM Trans. Database Systems*, Vol. 7, No. 3, Sept. 1982.
3. A. M. Keller, "Algorithms for Translating View Updates to Database Updates for Views Involving Selections, Projections, and Joins," *Proc. Fourth ACM Sigact-Sigmod Symp. Principles of Database Systems*, Mar. 1985.
4. "Final Report c: the ANSI/X3/SPARC DBS-SG Relational Database Task Group," M. Brodie and J. Schmidt, eds., in *Sigmod Record*, Vol. 12, No. 4, July 1982.
5. D. Maier, *Theory of Relational Databases*, Computer Science Press, Rockville, Md., 1983.
6. J. D. Ullman, *Principles of Database Systems*, 2nd ed., Computer Science Press, Rockville, Md., 1982.
7. A. M. Keller and J. D. Ullman, "On Complementary and Independent Mappings on Databases," *1984 ACM/Sigmod Int'l Conf. Management of Data*, Boston, Mass., June 1984.
8. S. S. Cosmadakis and C. H. Papadimitriou, "Updates of Relational Views," in *J. ACM*, Vol. 31, No. 4, Oct. 1984.
9. A. M. Keller, "Updates to Relational Databases Through Views Involving Joins," in *Improving Database Usability and Responsiveness*, P. Scheuermann, ed., Academic Press, New York, 1982.
10. A. L. Furtado and M. A. Casanova, "Updating Relational Views," in *Query Processing in Database Systems*, W. Kim, D. S. Reiner, and D. S. Batory, eds., Springer-Verlag, New York, 1985.
11. J. Davidson and S. J. Kaplan, "Natural Language Access to Databases: Interpretation of Update Requests," *Proc. Seventh Int'l Joint Conf. Artificial Intelligence*, Vancouver, B.C., Canada, Aug. 1981.
12. S. J. Kaplan and J. Davidson, "Interpreting Natural Language Database Updates," *Proc. 19th Annual Meeting of the Association for Computational Linguistics*, Stanford, Calif., June 1981.
13. J. D. Ullman, "The U.R. Strikes Back," in *Proc. ACM Symp. Principles of Database Systems*, Los Angeles, Mar. 1982.
14. S. J. Hegner, "Canonical View Update Support Through Boolean Algebras of Components," *Proc. Third ACM Sigact-Sigmod Symp. Principles of Database Systems*, Apr. 1984.
15. C. R. Carlson and A. K. Arora, "The Updatability of Relational Views Based on Functional Dependencies," *Third Int'l Computer Software and Applications Conf.*, (IEEE Computer Society), Chicago, Ill., Nov. 1979.
16. A. L. Furtado, K. C. Sevcik, and C. S. dos Santos, "Permitting Updates Through Views of Data Bases," *Information Systems*, Vol. 4, No. 4, Pergamon Press, Elmsford, N.Y., 1979.
17. E. K. Clemons, "An External Schema Facility to Support Data Base Updates," in *Databases: Improving Usability and Responsiveness*, Academic Press, New York, 1978.
18. K. C. Sevcik and A. L. Furtado, "Complete and Compatible Sets of Update Operators," *Proc. Int'l. Conf. Management of Data (ICMOD)*, Milan, Italy, June 1978.
19. L. Rowe and K. A. Schoens, "Data Abstractions, Views, and Updates in RIGEL," *Proc. ACM Sigmod Int'l Conf. Management of Data*, Boston, Mass., May 1979.
20. L. Tuchermann, A. L. Furtado, and M. A. Casanova, "A Pragmatic Approach to Structured Database Design," *Proc. Ninth VLDB Conf.*, Florence, Italy, Oct. 1983.
21. Y. Masunaga, "A Relational Database View Update Translation Mechanism," Report RJ3742, IBM San Jose Research Laboratory, San Jose, Calif., 1983.
22. G. Wiederhold, *Database Design*, 2nd ed., McGraw-Hill, New York, 1983.
23. P. Honeyman, "Extension Joins," *Proc. Int'l Conf. Very Large Data Bases*, Montreal, 1980.
24. A. M. Keller, "Choosing a View Update Translator by Dialog at View Definition Time," submitted for publication.
25. E. F. Codd, "Relational Database: A Practical Foundation for Productivity," *Comm. ACM*, Vol. 25, No. 2, Feb. 1982.

Arthur M. Keller is an assistant professor of computer sciences at the University of Texas at Austin. He has lectured widely on databases, distributed databases, and computerized typesetting. He is the author of *A First Course in Computer Programming Using Pascal* (McGraw-Hill, New York, 1982).

Keller received his PhD from Stanford University in 1985. His thesis was titled, "Updating Relational Databases Through Views." He received his MS from Stanford and his BS from Brooklyn College (CUNY).

Questions about this article can be directed to Keller at the University of Texas at Austin, Dept. of Computer Sciences, Austin, TX 78712-1188.